CMU 17-654 & 17-754 | Analysis of Software Artifacts | Spring 2006

Individual Project: Tool Analysis

May 18, 2006
Eun-young Cho
echo1@andrew.cmu.edu

JProfiler: Code Coverage Analysis Tool for OMP Project

Table of Contents

1. Project Objectives

To use JProfiler, an analysis tool, to report performance losses to:

- Report memory leaks on Nemo (an example of an Overlay Multicast Protocol, OMP)
- Resolve threading issues on Nemo
- Gain insight into the group management aspect of OMP using Nemo
- Use the gained knowledge for the MSE POSDATA studio project

2. Background

2.1 About JProfiler:

JProfiler is a unique tool when compared to any of its peers as:

- The tool uses a combined approach to provide different perspectives.
- The tool provides a faster 4 in 1 approach where the 4 views in one window correspond to Memory views, CPU Views, Thread Views and VM Telemetry Views.

The details of the aforementioned views are given below:

Memory Views

This view provides for:

- Heap walker styled drill down showing object references
- The drill down reports problem spots with a tree like representation of the Heap data structure.
- Detailed browsing of the Heap structure, in order to get information on memory and object references.

CPU Views

This view provides for:

- Showing threads information on invocation of threads and their back traces.
- Filtering mechanism enables the customizing the data at hand to one's own perspective
- A real time dynamic picture on the views

Thread Views

This view provides for:

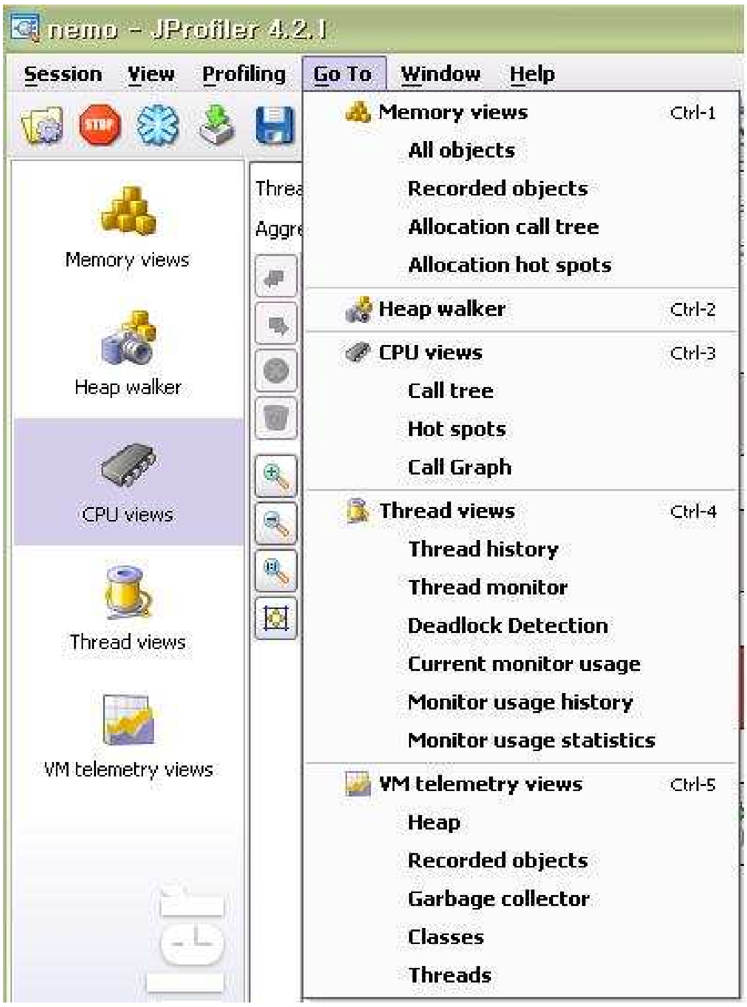
- Deadlock profiling by showing thread monitoring and colored coded thread history which enables programmers to catch deadlocks where they might potentially exist.
- The thread debugger is also included in the JProfiler's installation package.

VM Telemetry Views

This view provides for:

- Information on the Virtual machine's parameter from the moment the JVM starts.
- Each view breaks down into a sub view to enhance display and readability. This also makes the switching between views very easy and vastly possible.

The following snapshot shows JProfiler with its major views drop-down menu.



Followings are the comparison with similar tools in 2003. It's a competitive tool in perspective of features and costs. JProfiler got 2003 and 2005 Java Developer's Journal Readers' Choice Awards as the Best Java Profiling/Testing Tool. JProfiler was nominated and adjudged as the best Java Profiling tool for the 2003 and 2005 Java Developer's Journal Readers' Choice Award.

	Optimizeit Suite	JProbe Suite	JProfiler
Version	5.0	5.0	2.2.1
Price	$1,599	$2,000[1]	$499
Free evaluation	Yes	Yes	Yes
Online (built-in) help	Yes	Yes (JavaHelp)	Yes (JavaHelp)[2]
Is help context-sensitive?	Yes	Yes	Yes
Built-in tutorials	Yes	Yes	No[3]
Paper documentation	No	Yes	No
Number of tool modules	3 (Profiler, Thread Debugger, Code Coverage)	4 (Profiler, Coverage, Memory Debugger, Threadalyzer)	0 (all-in-one)
Tool modules sold separately?	No	Yes	No
CPU profiler	Yes (not real time)	Yes (not real time)	Yes (real time)
Object/heap profiler	Yes	Yes	Yes
Thread profiler	Yes	Yes	Yes
Deadlock detection	Automated and visual	Automated	Manual
Race condition detection	No	Yes	No
Code coverage	Yes	Yes	No
Multi-JVM support	Yes	Yes	Yes
Drill-down to source	Yes	Yes	Yes
Drill-down to bytecode	No	Yes	Yes
Remote profiling*	Yes	Yes	Yes
Automated profiling**	Yes	Yes	Yes
IDE integration	Yes	Yes	Yes
Report generation	Yes	Yes	Yes
Host platform licensing policy	Multiplatform and single-platform licenses	Single platform	Multiplatform
Website	www.borland.com/optimizeit	www.jprobe.com	www.jprofiler.com
Ease of use	7/10	4/10	8/10

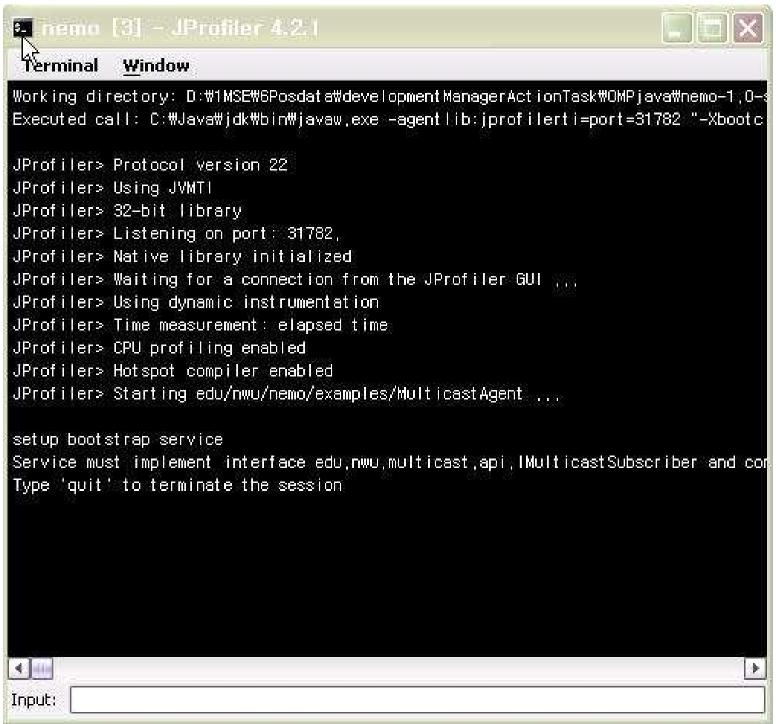
Source: http://www.javaworld.com/javaworld/jw-08-2003/jw-0822-profiler.html

* Remote profiling: The ability to profile a Java program executing on a machine other than your development machine
** Automated profiling: The ability to perform unattended overnight profiling sessions; in other words, command-line-driven operation with no GUI
[1] JProbe Suite price includes one year of Gold Support (technical support)
[2] ej-technologies' JProfiler Online Help contains almost no screenshots of views or dialogs
[3] ej-technologies' lack of explicit tutorials is partly compensated by some demo sessions

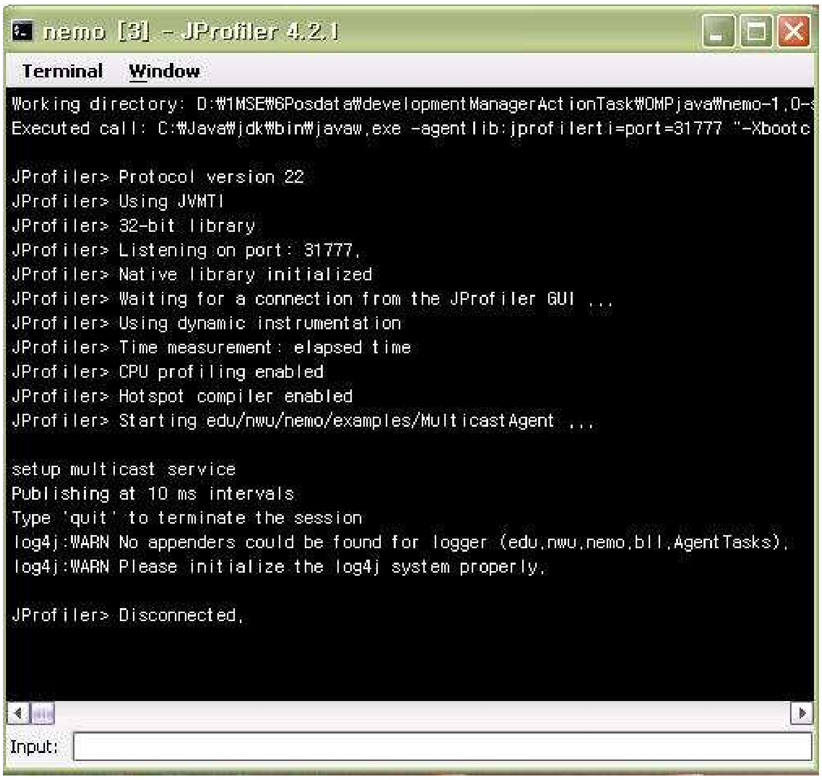
This is particularly of interest because on POSDATA, SI Company in Korea, my studio project requires the use of OMP in order to broadcast video stream to particular nodes through the use of group management. Though the focus of the JProfiler testing is not for OMP in general, the program of interest is Nemo, a multicast protocol that uses group management. Here Nemo is an existing OMP project that serves a good example to show the kind of problems and parameters one may have to face or think about when one is realizing an OMP. The main types of nodes are:

- **Bootstrap Node**: Nodes that serve as the leader of a cluster or a segment of a group. They serve as the meeting point for publishers and subscribers
- **Publisher Nodes**: Publish data and send data to subscribers. They may also be known as co-leaders in the OMP terminology. The co-leader is like a server for a particular layer consisting of OMP clients.
- **Subscriber Nodes**: Receive data from publishers. They can be referred to as clients.

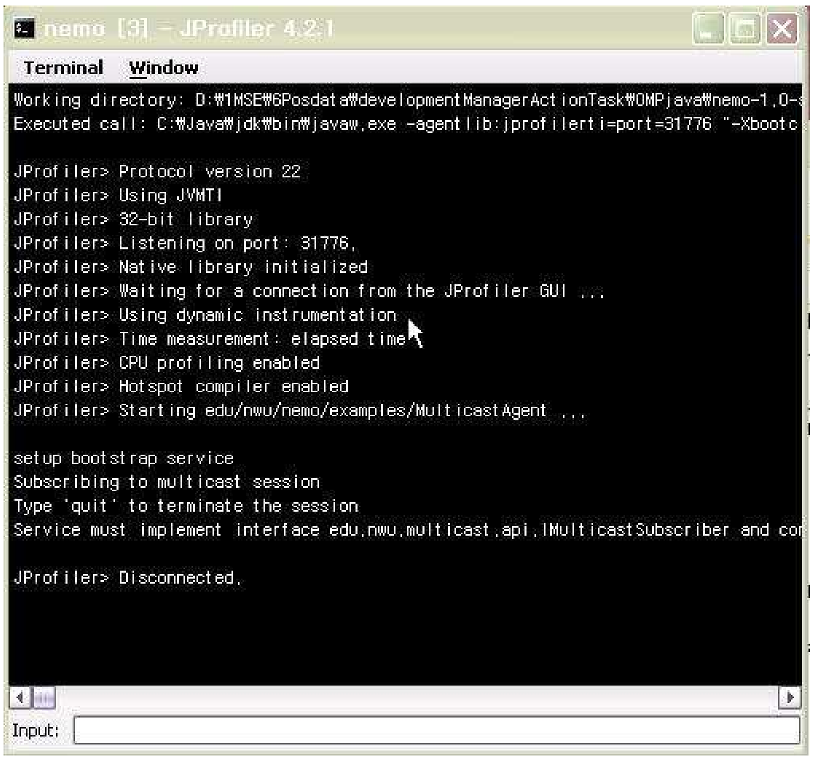
Followings are the script in case of running for each node role.



(a) Bootstrap node



(b) Publisher Node



(c) Subscriber Node

Figure 1: Execution script using the JProfiler tool

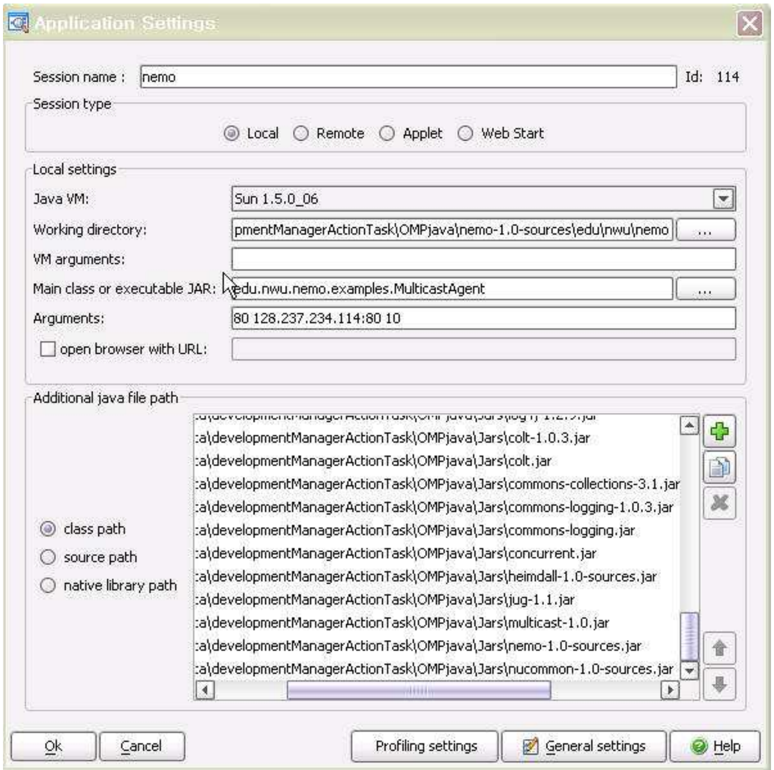
2.2 About Nemo and JProfiler's scope on Nemo

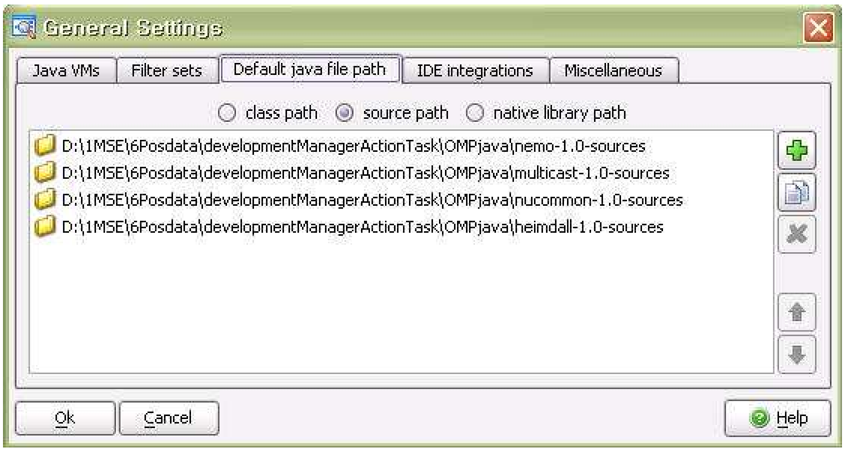
Nemo, an open source project, implements the concept of Overlay Multicast Protocol which is a networking protocol to share a single data stream between a large number of connecting clients without degradation of the performance or increase in network cost. In the present situation on the MSE project, there is a requirement to deal with the high degree of variability that may exist in the network on account of the join and leave operations of the clients. This variability arises from the dynamic situation of a large number of nodes joining and leaving, the network. The aim of any multicast protocol is to achieve this variability without giving up on end to end delays and providing for an additional network costs.

3. Experimental Setup

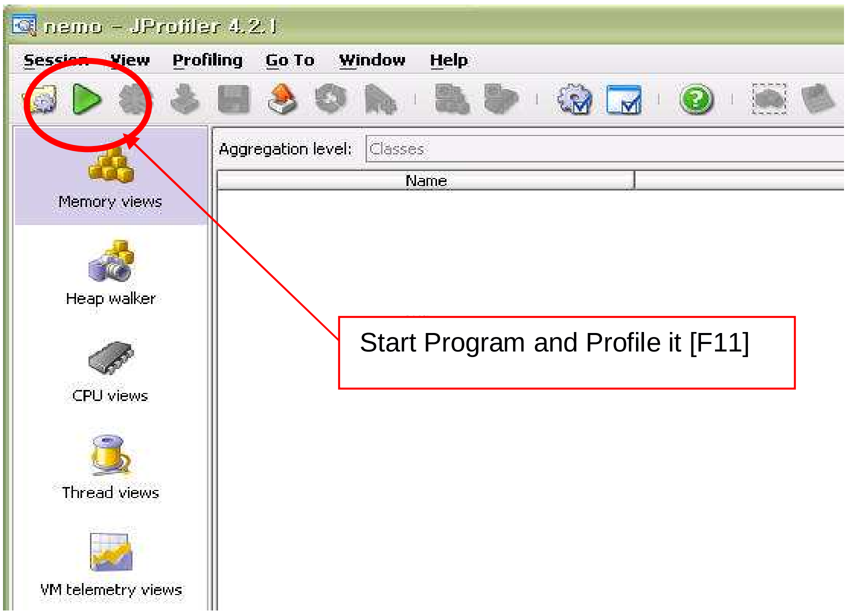
3.1 JProfiler Installation and Setup

In order to run JProfiler, several steps should be done. First, download the evaluation version (JProfiler 4.2.1) of JProfiler which is jprofiler_windows_4_2_1.exe in the website, http://www.ej-technologies.com/download/jprofiler/trial.php. Second, install JProfiler. Finally, run JProfiler with personal evaluation key which received via email. For windows user below environment setup is needed.



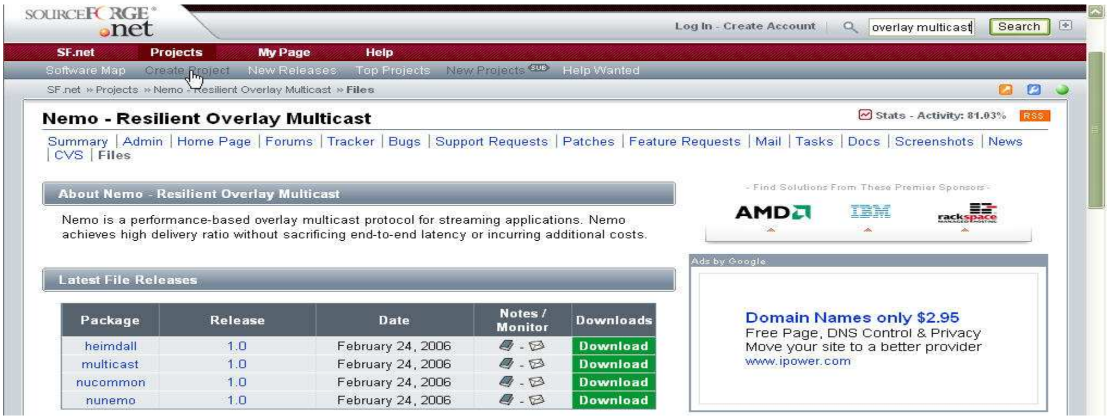


After successful setting of the target programs, a program can be monitored by JProfiler in four areas.



3.2. Nemo Installation

Nemo is an open-source overlay multicast protocol for streaming applications provided by Northwestern University. In order to run Nemo, Nemo source file and additional jar files are need to Download. The source website is as follows.
http://www.aqualab.cs.northwestern.edu/projects/nemo/download.php or
http://sourceforge.net/project/showfiles.php?group_id=160473



3.3. Nemo Execution

Nemo provides sample program named MulticastAgent.java to test overlay multicast protocol. The program can be run using three different set of parameters. When it runs using one parameter which is port number, the agent program runs as a bootstrap. A subscriber needs one more parameter, the address of bootstrap agent. A publisher needs additional packet sending interval. In this experiment, I start using parameters in the MulticastAgent.java as follows.

Node type	**Input Parameter**	**Meaning**
Bootstrap	80	Local Port Number
Subscriber	80 128.237.234.114:80	Port No., Bootstrap IP + Port No.
Publisher	80 128.237.234.114:80 10	Port No., Bootstrap IP + Port No., Publish-Rate

4. Analysis of Results

4.1 Memory View through Nemo

As mentioned before, several memory views support heap analysis. Following snapshot shows the class monitor subview.

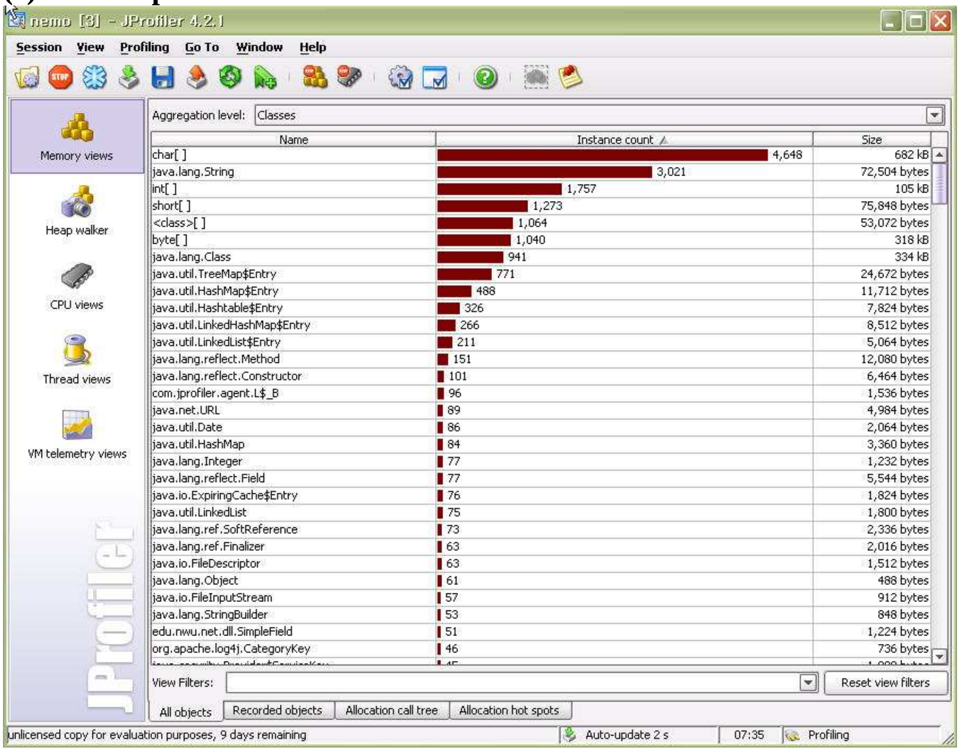
As can be seen through the memory view, the initial large allocation is for the logging feature provided inbuilt into Nemo. This feature is commonly found on all the three node types. The next large memory allocations are for the receive queue and the packet socket. When compared at the method aggregation level, the Nemo BootstrapService.setup is the class with maximum allocation to its methods.

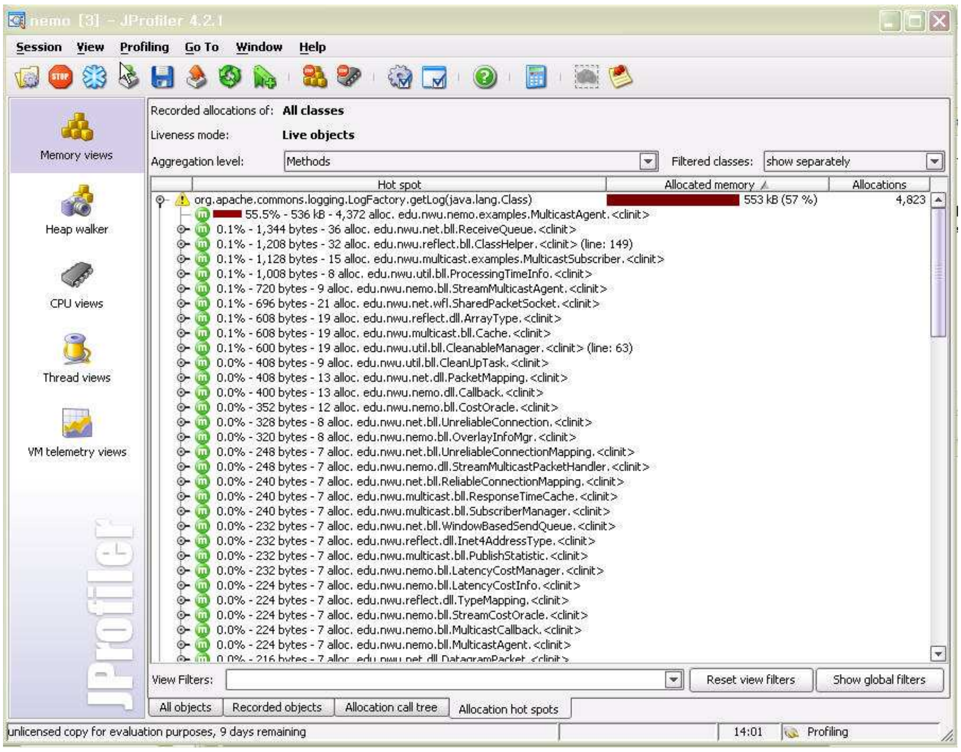
In following memory views, initial large memory allocation is for the log. It's common on three node types. And next common allocations are for the ReceiveQueue and PacketSocket.

In method aggregation level, NemoBootstrapService.setup is major allocated class and its method.

For the bootstrap mode the view is defined below:

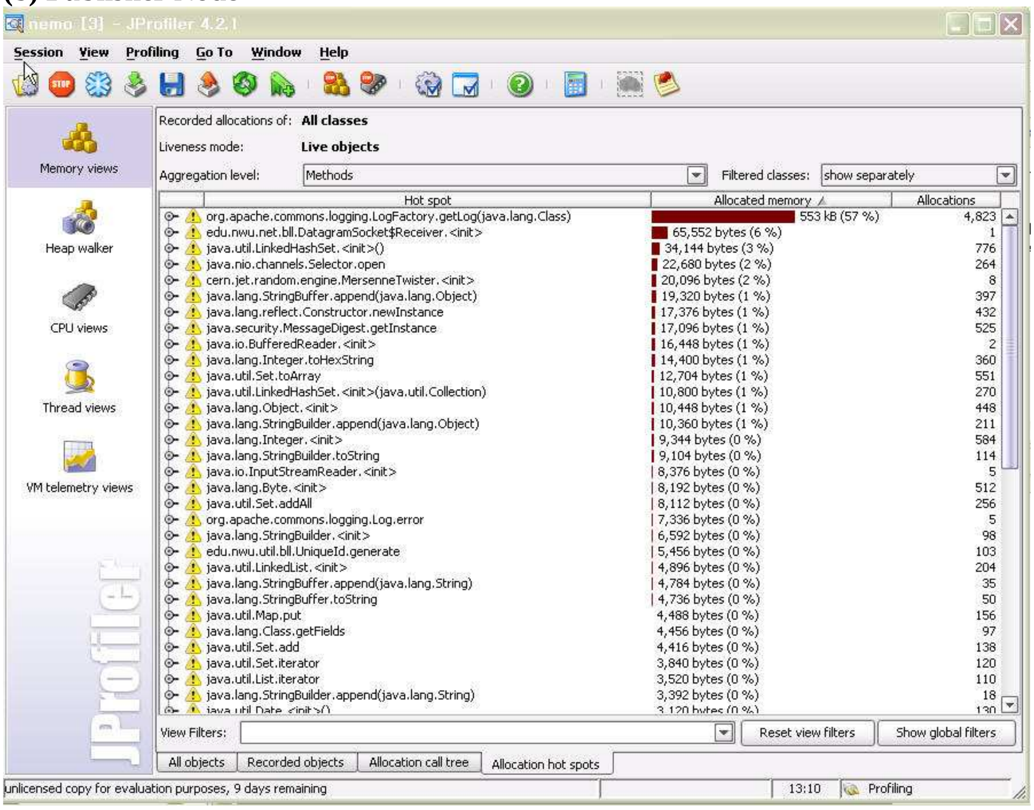
(a) Bootstrap Node

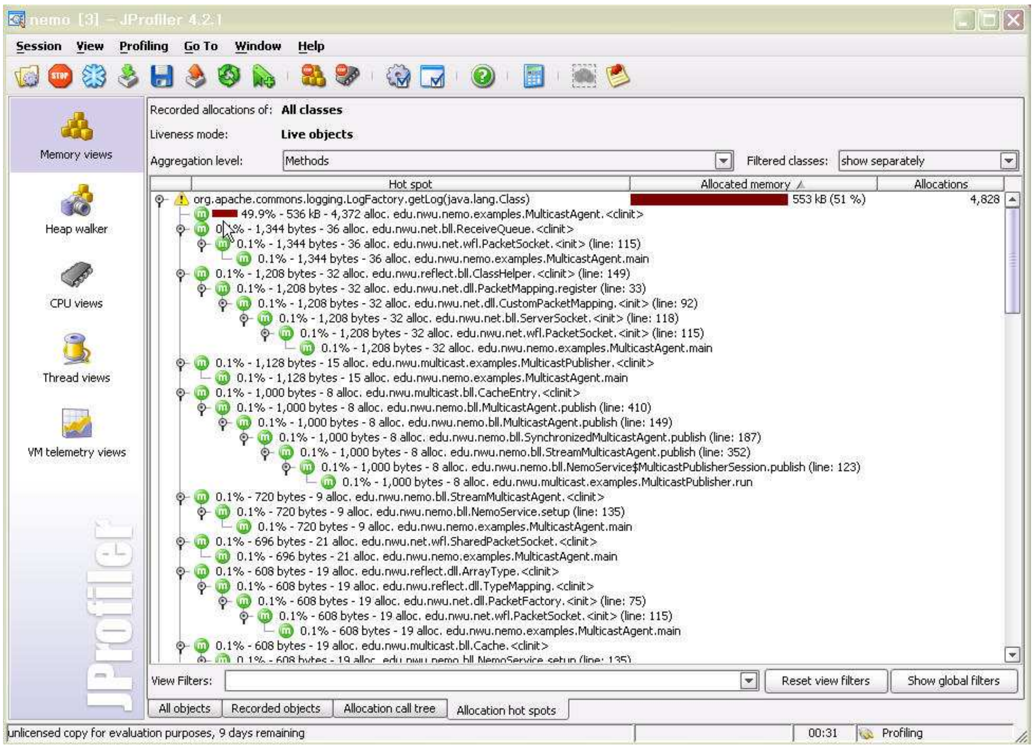




For the publisher node the view is defined below:

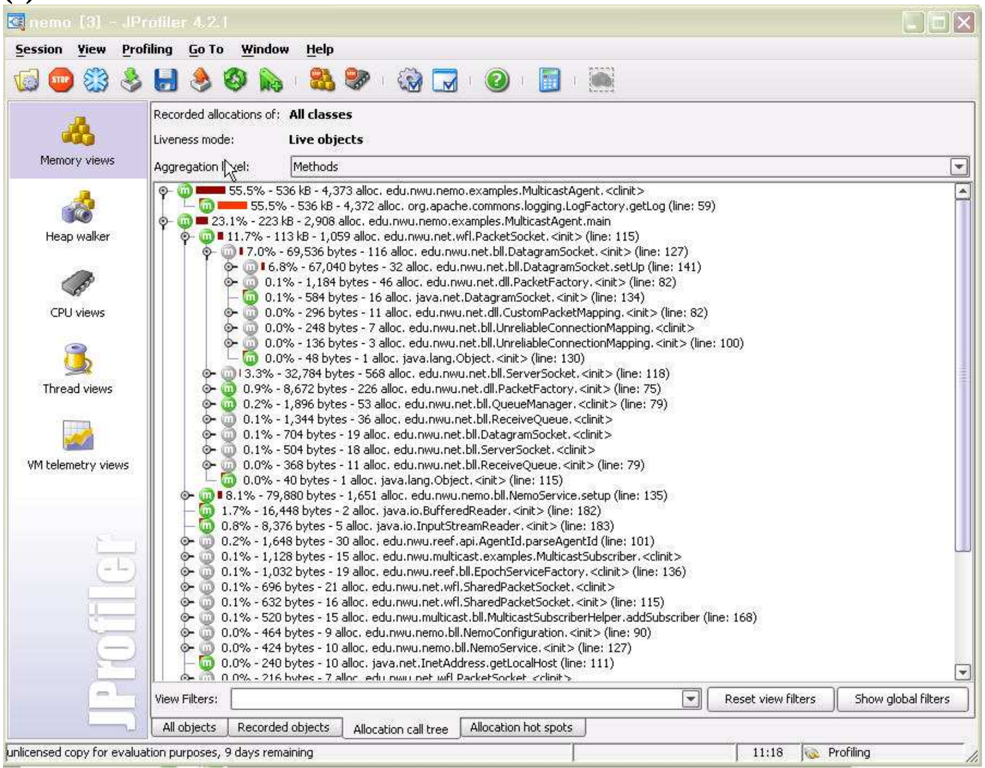
(b) Publisher Node

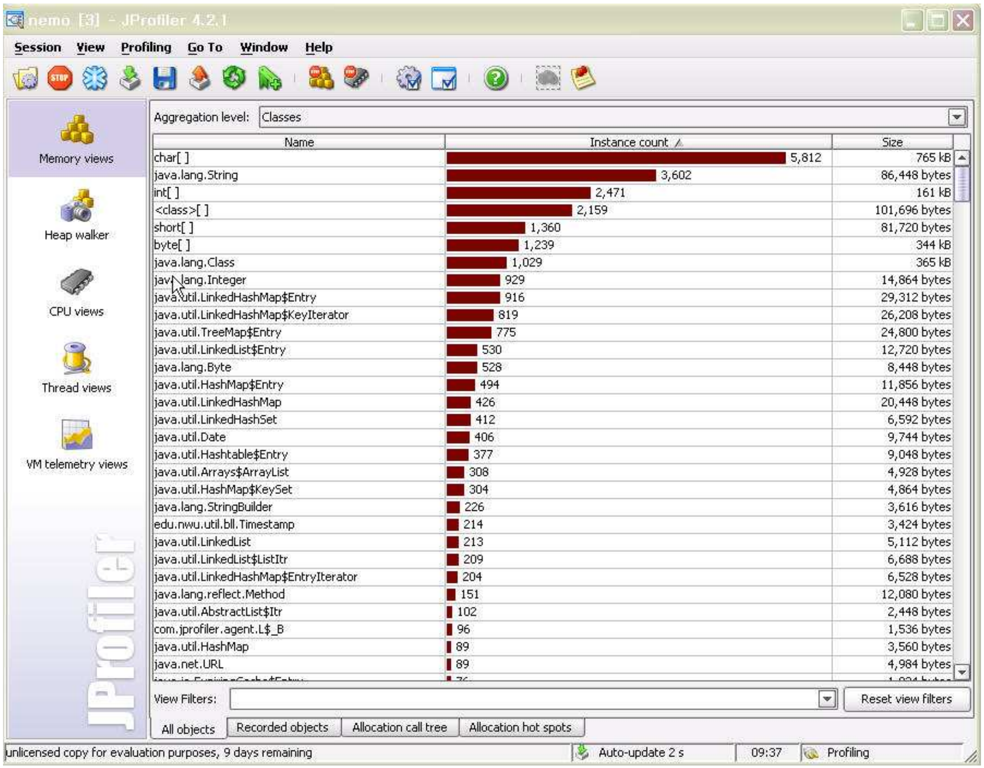




For the subscriber node the view is defined below:

(c) Subscriber Node





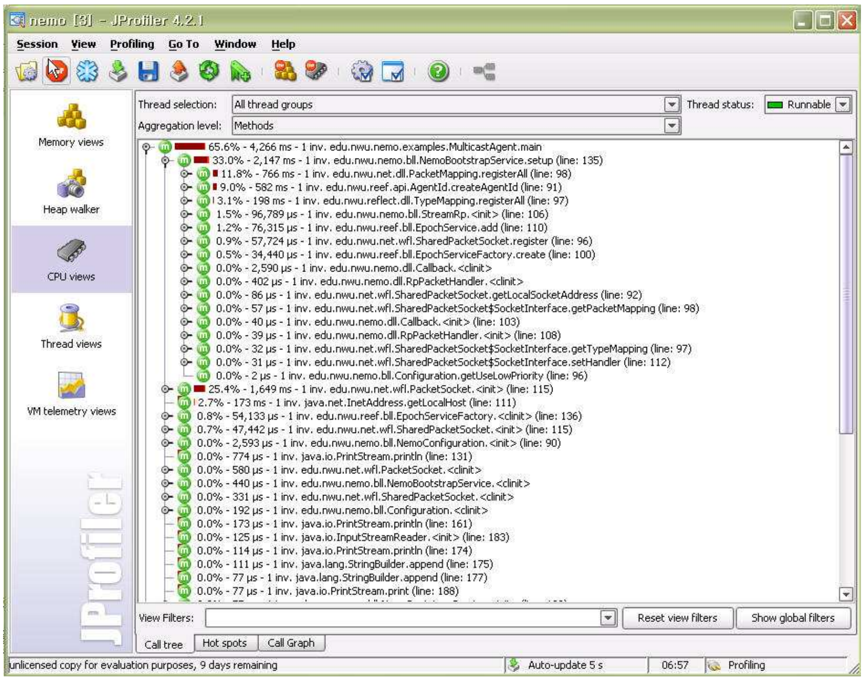
4.2 CPU View through Nemo

Method timing is available as a method invocation tree showing percentage of time consumed and absolute time consumed. Following snapshot shows a view of such a tree.

In following given CPU view, again an initial large memory allocation is for the logging feature inbuilt in Nemo This feature is commonly found on all the three node types. The next large memory allocations are for the receive queue and the packet socket. When compared at the method aggregation level, the Nemo BootstrapService.setup is the class with maximum allocation to its methods.

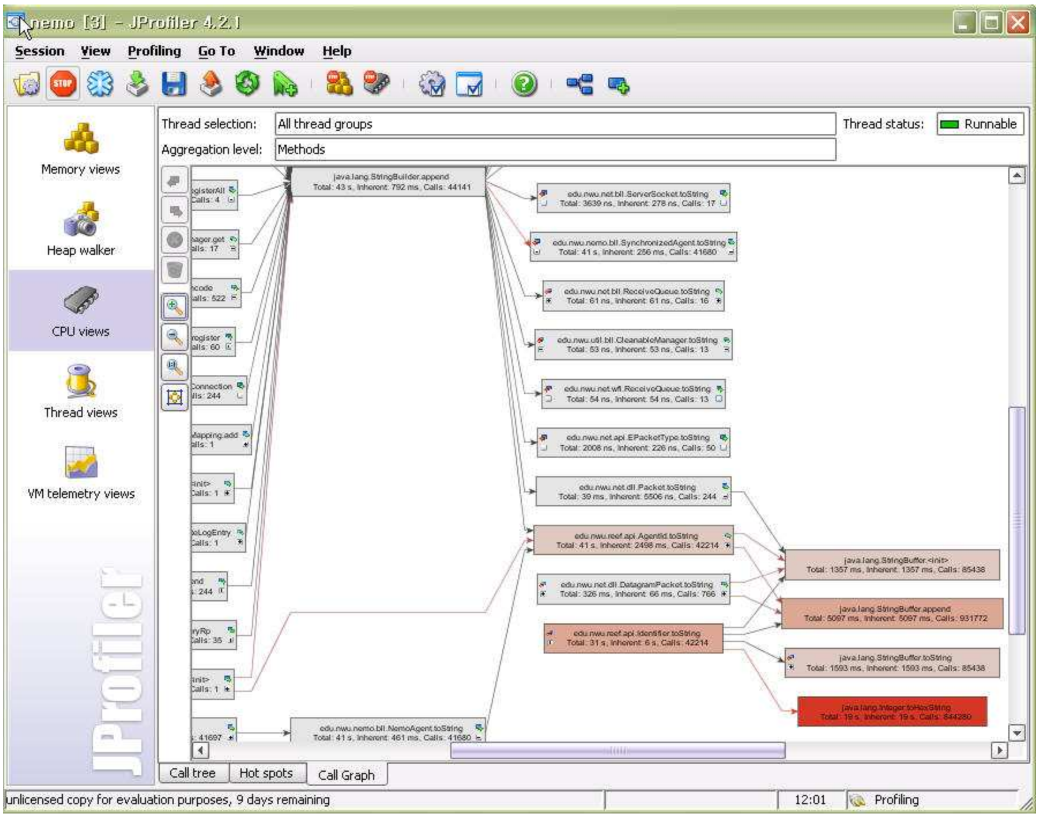
In Publisher node, StreamMulticastAgent.publish is the class with the next dominant allocation. Finally, hot spot in Subscriber type node is allocated for the sendPkt. The next consumption is for the LogFactory.getLog class.

(a) Bootstrap Node

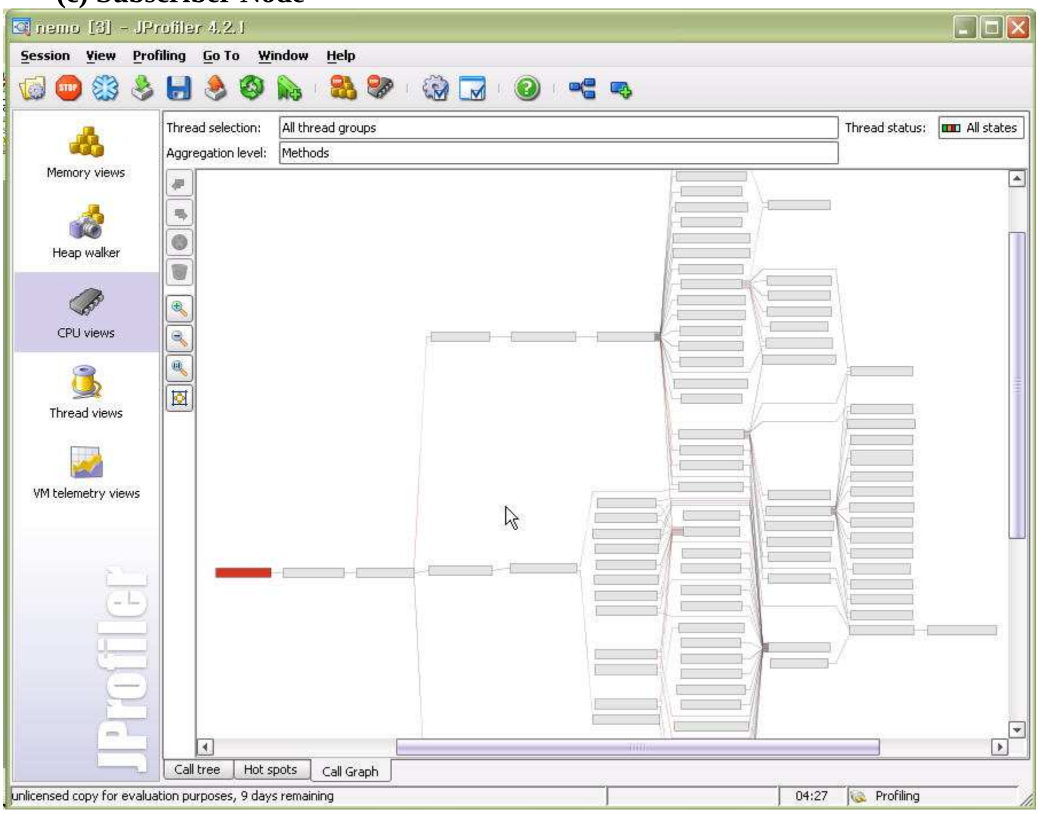


(b) Publisher Node

The following snapshot shows a statically calculated thread resolved call graph from main function which is selected in graph nodes. The graph nodes can be methods, classes, packages, or J2EE components, depending on the selected aggregation level. If a graph has been calculated, the context menu also provides access to this action. The resulting graph is static and can be re-calculated be executing Generate graph again[5]. The call graph wizard remembers the last selection. Using this graph, the caller-callee relationship is visibly analyzed. The node color is marked from a gray to red scale for increasing the inherent time and the total time. Therefore, it becomes possible to identify the potential bottlenecks.



(c) Subscriber Node

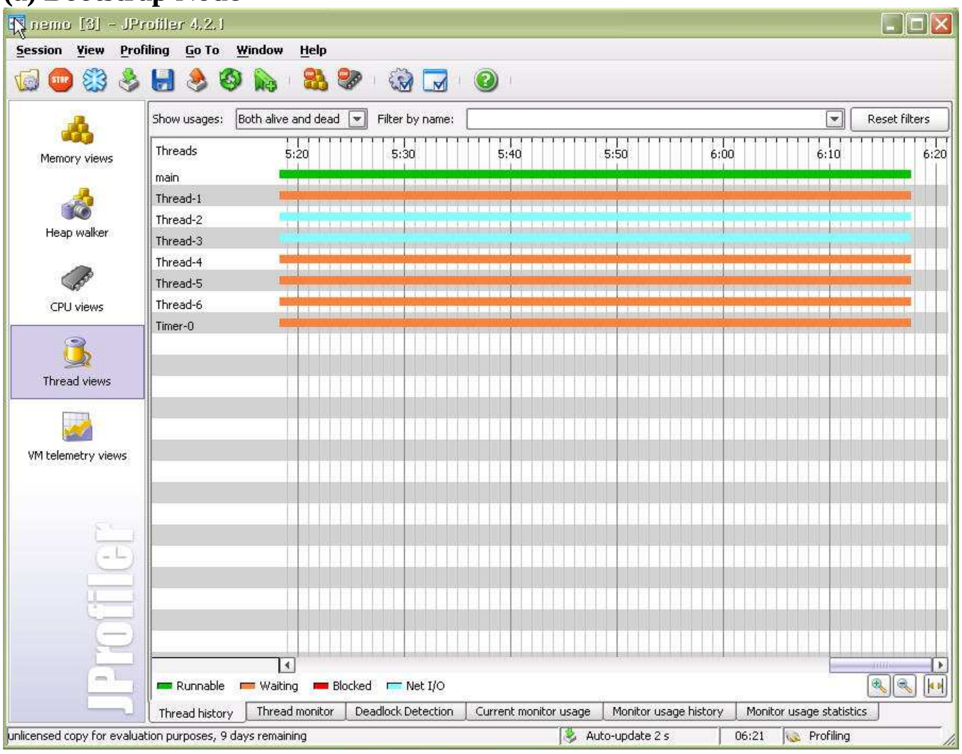


4.3 Thread View through Nemo

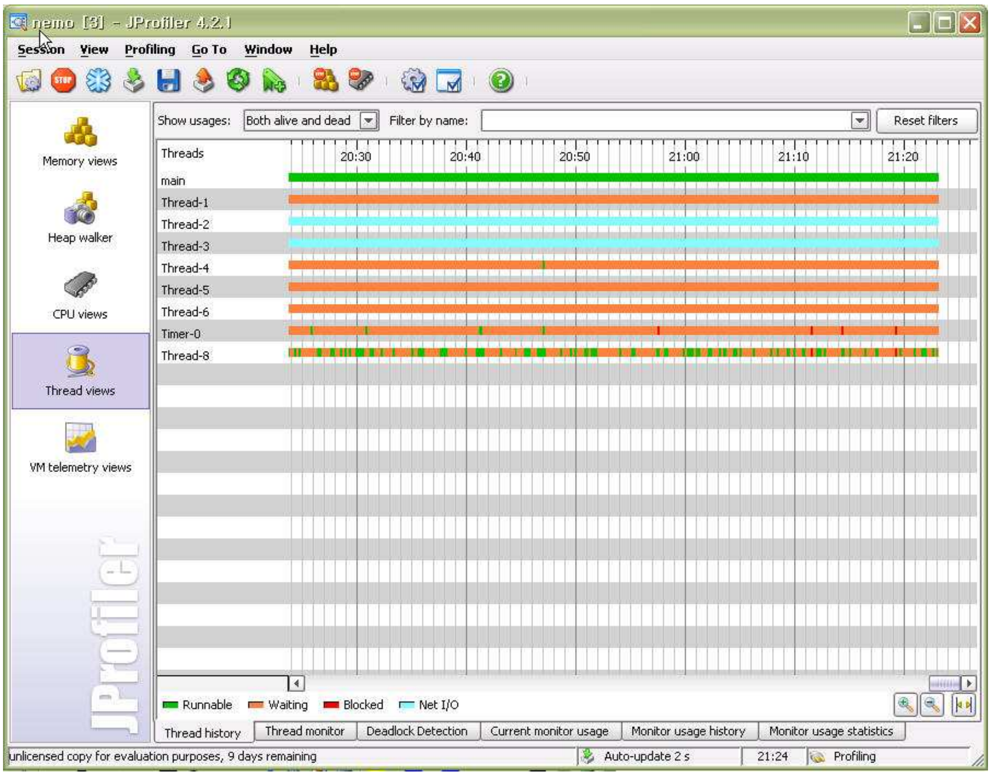
The threads view set comprises five subviews focusing on past and current thread states (the color scheme of these is as follows: runnable-green, waiting-orange, net I/O-light blue, and blocked-red), past and current monitor usage, and monitor statistics [5]. Following figures show a typical threads view.

The pattern of three nodes are similar to each other with an exception in the case of the thread of publish. There is a provision for an additional thread for the publishing function. In case of short interval, the thread has more control time which is represented by the green color. Based on this view, the thread bottleneck detection is easily identified. As can bee seen the thread for publishing node is in "waiting" state for a longer duration when the interval gets longer. Performance tuning becomes simpler using this view.

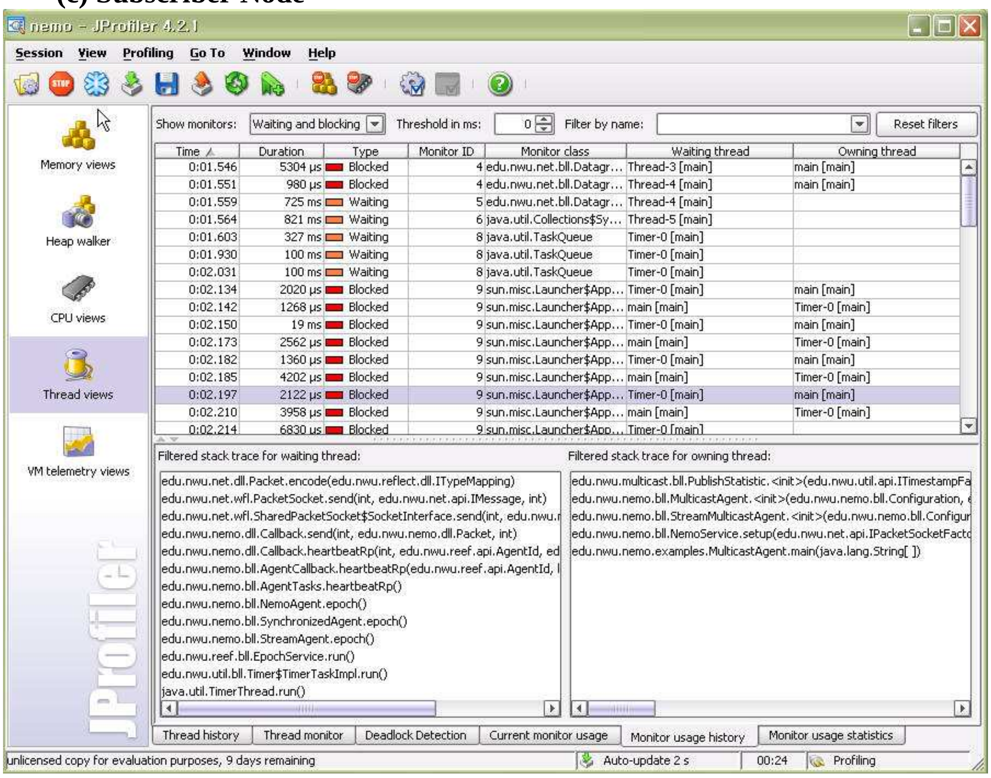
(a) Bootstrap Node



(b) Publisher Node



(c) Subscriber Node

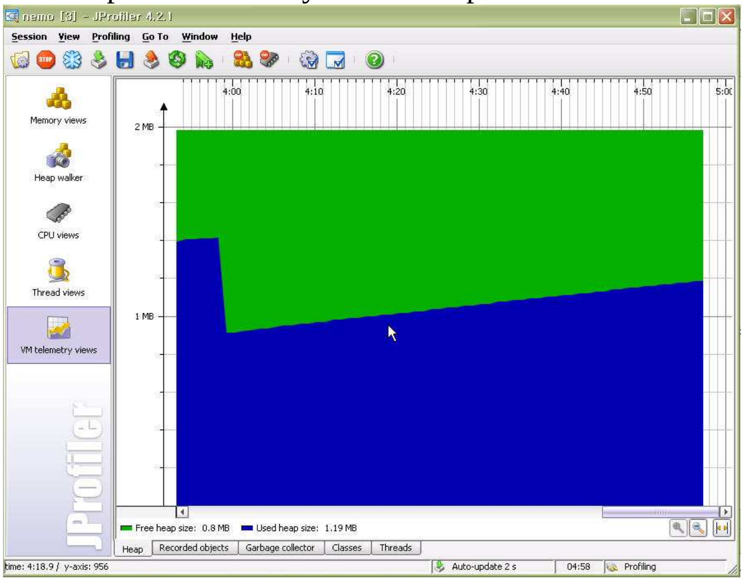


4.4 VMtelemetry View through Nemo

The VM telemetry view set comprises five different real-time scrolling graphs showing used and free heap space, number of objects (helpfully categorized into arrays and non-arrays), number of loaded classes, garbage collector activity, and number of threads [5]. Following snapshots show this view set.

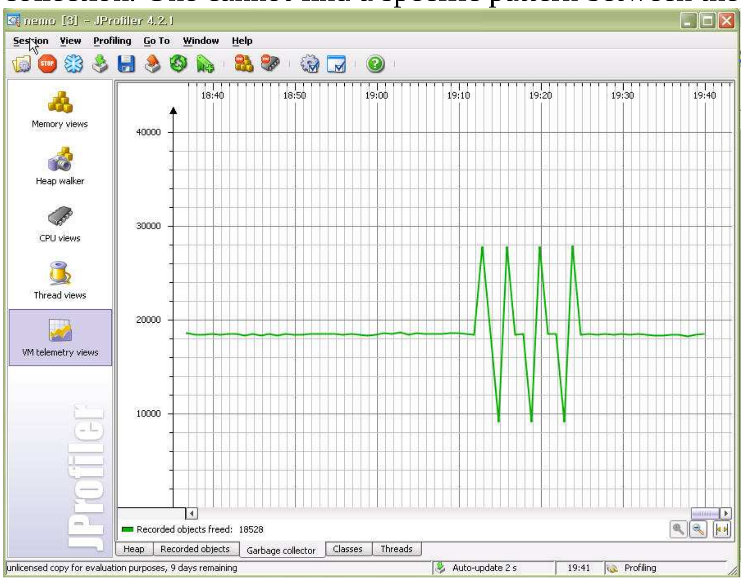
(a) Bootstrap Node

This graph shows the assigned and freed heap under total 1.98Mbytes. The status of bootstrap node is less dynamic than publisher node.



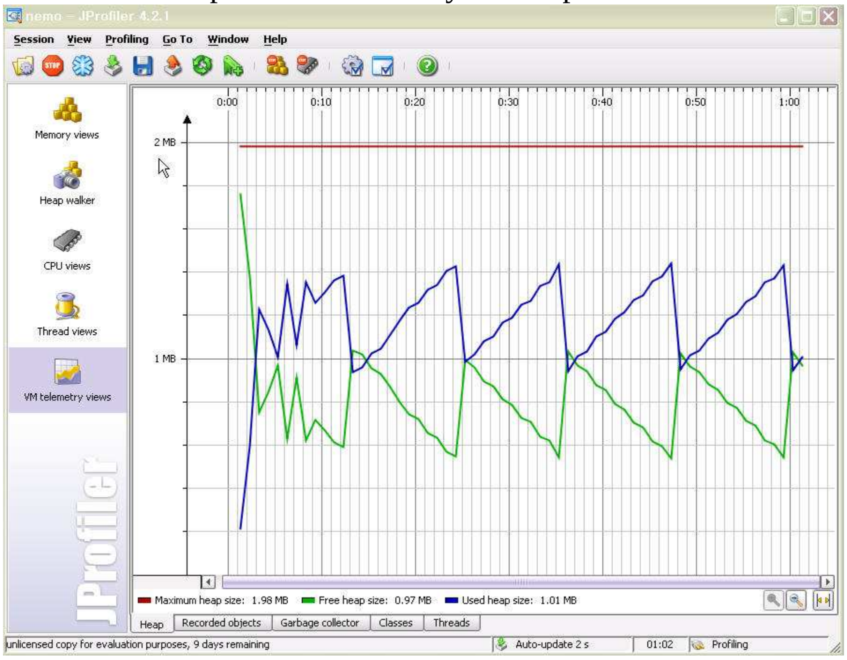
(b) Publisher Node

This snapshot provides an analysis of the freed objects from the garbage collection function. During the course of object creation and deletion, the shape of graph becomes fluctuating. However, in case the interaction is low there is probably a little of garbage collection. One cannot find a specific pattern between the node types.

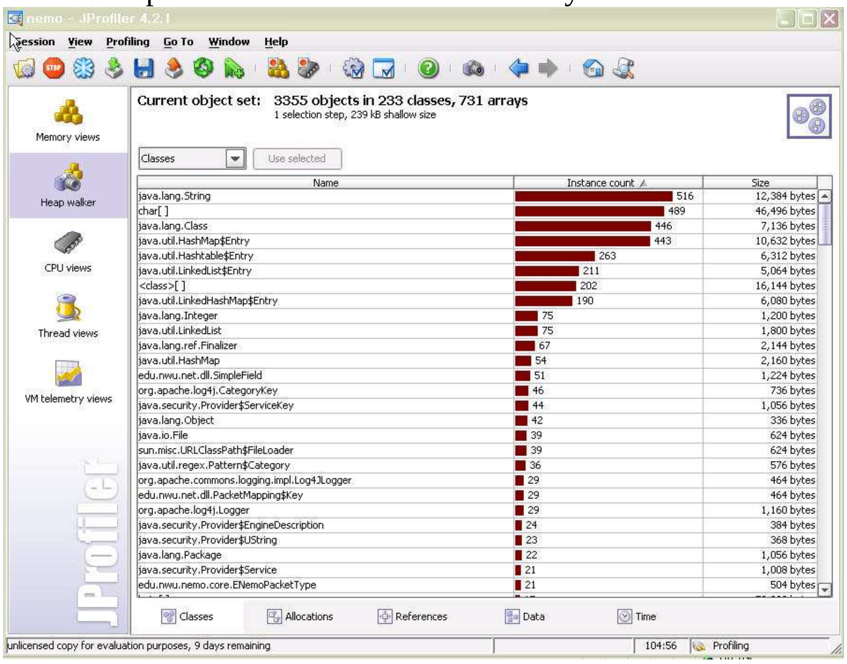


(d) Subscriber Node

The heap status view provides information on the allocated and freed space from a total of 1.98 MB. This space is used mainly for the periodical socket communication.



Lastly, JProfiler's Heap Walker module is the unique aspect based on the easy-to-navigate GUI [5]. The following snapshot view shows the status of classes and arrays in bootstrap nodes. The developer follows the source code easily in run time.



5. Lessons Learned

Throughout the project, I learned a number of lessons by using JProfiler on an OMP project like Nemo. JProfiler's various output produced enabled to get knowledge on Nemo's setup as an OMP project and also form an opinion on whether Nemo would provide useful support to my team project.

5.1 General Characteristics of JProfiler

There are several profiling tools available on the market, such as JProbe and Optimizeit, and to be fair, they all kind of do the same thing. The core features that most end-users are ingested in are the same as other tools - thread monitoring, deadlock detection and memory/class instance monitoring.

JProfiler provides the obvious two advantages as follows.

- Easy to use: The main window is simple and intuitive, allowing a user to quickly navigate between the different views on offer. Filters are also very straightforward and provide a way to focus on a specific set of information.
- Cost: It's one of the 500$ java products.

Ant it also runs on Mac OS X and provides easy application server integration.

.

5.2 Benefits to the MSE studio project

1. JProfiler 2.4 is designed to help developers manage performance risks throughout the development process and produce fast, reliable enterprise applications. Developers use profiling technology to identify performance bottlenecks and memory leaks during the development stage of an application. JProfiler is the enterprise-level Java tool available to the development community that integrates CPU, memory and thread profiling in one powerful and robust application.

2. Based on the expectations and the results produced by JProfiler, the analysis proved to be beneficial in understanding a simple OMP situation. The same situation was used to make decisions of making Nemo a part of the MSE project.

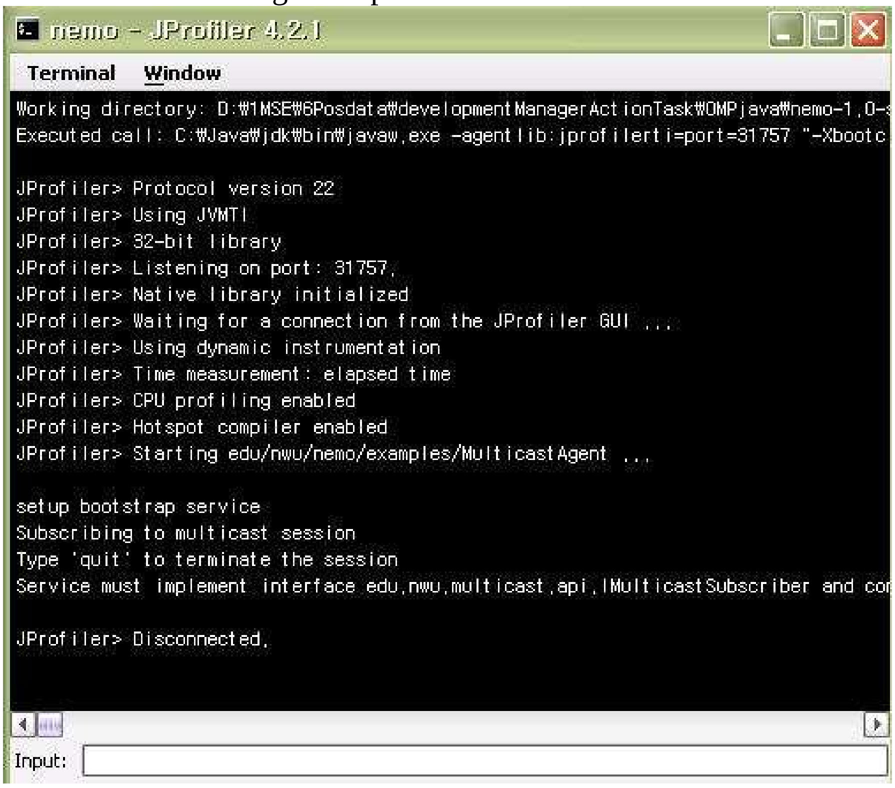
3. The analysis are primarily of 4 views:
    - **Memory View**: JProfiler's memory view section offers dynamically updated views on memory usage and allocations. All views can show live and garbage collected objects.
    - **CPU View**: JProfiler offers various ways to record the call tree to optimize for performance or detail. The thread or thread group as well as the thread status can be chosen for all views.

- **Thread View**: For thread profiling, JProfiler offers thread history, thread monitor, deadlock detection graph, current monitor usage, monitor usage history, and monitor usage statistics.
- **VM Telemetry View**: To observe the internal state of your JVM, JProfiler offers various telemetry views such as heap, objects, garbage collector, classes and threads.

These profiles provide useful information as described above. For instance, the Logging function is an inherent scheme available in Nemo. Even though the overall performance of Nemo is better than a previous prototype example in terms of NICE (another OMP protocol), it is still a bottleneck in Nemo. Therefore, initial logging function should be designed as an optional function. In conclusion, the one suggestion is that the logging function is replaced by an Aspect-Oriented Program (AOP) which depends upon the analysis result of JProfiler.

5.3 Drawbacks

1. JProfiler does not provide control over selectively profiling java code in fine-grained level.
2. There is no the facility to extract information from the response of a request and use that information in subsequent requests.
3. Even though some of multiple executions are done, the disconnected shell script window console still remained like a dangling program as the following snapshot. Therefore, as an improvement step, the related console should disappear when the execution is done and the tool is no longer in operation.



6. Conclusions

In this approach, JProfiler as a static analysis tool for OMP project was moderately helpful. The reasons are as follows:

1. The tool could provide memory, CPU, and thread view for OMP open product, Nemo. This was when there is a bootstrap, publisher, and subscriber node in the network. This fact is attributed to the real time nature of JProfiler.
2. The tool could execute a simple case of an OMP on Nemo and provide insight into the network parameters.
3. The various outputs produced with JProfiler enabled to understand the OMP concepts as depicted by Nemo in a better and more productive way. This provides the useful idea on a new design and implementation of OMP in perspectives of time, space and also its related tradeoffs.
4. For the short project, the reverse engineering is very important to achieve the basic concept using previous academic or industry product. On that point, this simple and fast usable tool contributes to save the time and money of the project. JProfiler is strongly recommended to the urgent and similar size project.

7. References

1. http://www.ej-technologies.com/products/jprofiler/tutorials.html
2. http://www.javaworld.com/javaworld/jw-08-2003/jw-0822-profiler.html
3. http://sourceforge.net/project/showfiles.php?group_id=160473
4. http://weblogs.java.net/blog/simongbrown/archive/2005/02/jprofiler_minir_1.html
5. JProfiler Manual, ej-technologies, 2006
6. CMU MSE Team OMPArchitectability, Mini-Project 1: Tool or Analysis Practicum, Daikon: Invariant Detection of Nemo in association with OMP Project, Spring semester 2006